# Floating-Point and Complex Arithmetic Enhancements

**WG14/N964 J11/97-057 (Draft 1997-05-23)**

Jim Thomas
jthomas@best.com

This documents describes potential C9X enhancements related to the floating-point and complex arithmetic specifications. Section numbers refer to C9X Draft 9. Each enhancement could be accepted or rejected independently.

**1. Enhancement:** Clean up access to imaginary types:

a) Add a new <complex.h> macro `_Complex_I` which gives a complex representation of the imaginary unit *i* (even on implementations supporting imaginary types).

b) Require the macro `_Imaginary_I` to be an imaginary representation of *i* and specify it to be defined if and only if the implementation provides imaginary types as specified in G.3).

c) Allow the implementation to define the macro `I` to be `_Complex_I` or `_Imaginary_I`.

d) Reserve the token `imaginary` to be a keyword just when <complex.h> is included and `_Imaginary_I` is defined.

e) Add in normative text a conditionally defined macro signifying adherence to Annex G, while noting the annex's *informative* status.

f) Modify the definition of multiplication and division of two imaginary operands so that the result type is real.

**Rationale:** (a)-(c) give the programmer on systems with imaginary types the facility to control whether use of the macro `I` introduces an imaginary type. Disallowing imaginary types is useful for some programs intended to run on implementations without imaginary types.

(b) provides a test for whether imaginary types are supported (whether or not the implementation fully supports Annex G).

(b) and (d) reserve the keyword `imaginary` so that implementations can support Annex G without breaking conformance to the normative standard. An implementation that does not support imaginary types need not reserve the keyword `imaginary`.

(e) gives the programmer a way of testing whether the implementation claims support for Annex G, albeit without normative clout.

(f) provides a more natural behavior for imaginary types, while (a)-(c) provide an alternative for strict compatibility with systems without imaginary types.

**Edits:** *In 7.8, replace paragraphs [#2] and [#3] with:*

The macro

```
_Complex_I
```

expands to an expression of type `const float complex`, with the value of the imaginary unit[1]. The macro

```
_Imaginary_I
```

is defined if and only if the implementation supports imaginary types as specified in G.3; it expands to an expression of type `const float imaginary`, with the value of the imaginary unit. The macros are suitable for use in constant expressions.

The macro

```
I
```

is defined to be `_Complex_I` or, if defined, `_Imaginary_I`. Notwithstanding the provisions of subclause 7.1.3, it is permitted to undefine the macro `I`.

*In 6.1.1, add the keyword* `imaginary`.

*In 6.1.1, replace [2] with:*

The token `complex` is reserved in translation units where the header `<complex.h>` is included; the token `imaginary` is reserved in translation units where both the header `<complex.h>` is included and also the macro `_Imaginary_I` is defined; all other keyword tokens are reserved in all translation units. When reserved, the above tokens (entirely in lower case) are keywords (in translation phases 7 and 8), and shall not be used otherwise. When the token `complex` or `imaginary` is reserved, its use prior to the first inclusion of the header `<complex.h>` results in undefined behavior.

*In 6.8.8, replace paragraph [#2] with:*

The following macro names are conditionally defined by the implementation:

| | |
|---|---|
| `__STDC_IEC_559__` | The decimal constant 1, intended to indicate conformance to the specification in Annex F (IEC 559 floating-point arithmetic). |
| `__STDC_IEC_559_COMPLEX__` | The decimal constant 1, intended to indicate adherence to the specification in informative Annex G (IEC 559 compatible complex arithmetic). |

*In G.1, remove the sentence clause "*the macro expands to the decimal constant 1*".*

*Remove G.2 (and renumber subsequent subclauses).*

[1] The imaginary unit is a number $i$ such that $i*i = -1$.

*Replace the first paragraph of G.6 (prior to renumbering) with:*

The macro

```
_Imaginary_I
```

is defined, and the macro

```
I
```

is defined to be `_Imaginary_I` (7.8).

*In G.5.1 (prior to renumbering), paragraph [#1], replace the second sentence with:* If both operands have imaginary type, then the result has real type.

**2. Enhancement:** Change the type for the `INFINITY` and `NAN` macros from `float_t` to `float`.

**Rationale:** Currently these macros have different types depending on the evaluation method. This makes writing portable code unnecessarily difficult, as type-dependent use, e.g. `printf("%e", INFINITY)`, must be avoided or put under a switch. This proposal is in line with the earlier change in the corresponding real type of the imaginary unit `I` from `float_t` to `float`.

**Edits:** *In 7.7, paragraphs [#4] and [#5], replace the four occurrences of* `float_t` *with* `float`.

**3. Enhancement:** Annex F identifies the transformation `0 * x` --> `0.0` as invalid if `x` might be infinite or NaN. Add -0 to the invalidating values.

**Rationale:** This points out a case that the implementor of the optimizer might overlook.

**Edits:** *In F.8.2, paragraph [#1], bullet 0 * x --> 0.0, replace* "if `x` is a NaN or infinite" *with* "if `x` is a NaN, infinite, or -0".

**4. Enhancement:** Provide a function for *floating multiply-add*: `fma(x, y, z)` computes `x * y + z` as one ternary operation (i.e. with just one rounding). IEC 559 implementations must deliver the result with just one IEC 559-style rounding.

**Rationale:** In many cases, clever use of floating multiply-add leads to much improved code. However, its unexpected use by the compiler can undermine carefully written code. The `FP_CONTRACT` directive in the current floating-point spec can be used to disallow use of floating multiply-add. What's still needed is a way to guarantee its use just in particular situations. Many current machines provide hardware floating multiply-add instructions; software implementation can be used for others.

There was a suggestion to add a macro for testing whether, in effect, `fma` is implemented in hardware. Although potentially useful, such a macro is not being proposed here. Similar macros would seem to be needed for other functions and operations, notably the divide operator, and the need in the case of these existing routines has not been established.

*Insert the following subclause in 7.7 in the <math.h> section:*

## 7.7.y Floating multiply-add

### 7.7.y.1 The `fma` function

**Synopsis**

```
#include <math.h>
double fma(double x, double y, double z);
```

**Description**

The **fma** function computes the sum **z** plus the product **x** times **y**, rounded as one ternary operation: it computes the sum **z** plus the product **x** times **y** (as if) to infinite precision and rounds once to the result format, according to the rounding mode characterized by the value of **FLT_ROUNDS**.

**Returns**

The **fma** function returns the sum **z** plus the product **x** times **y**, rounded as one ternary operation.

[A longer alternative name might be **fmuladd**.]

*Insert the following subclause in F.9 of the IEC 559 annex:*

## F.9.y Floating multiply-add

### F.9.y.1 The `fma` function

**fma(x, y, z)** computes the sum **z** plus the product **x** times **y**, correctly rounded once.

- **fma(x, y, z)** returns one of its NaN arguments and raises no exception if **x** or **y** is a NaN.
- **fma(x, y, z)** returns a NaN and optionally raises the invalid exception if one of **x** and **y** is infinite, the other is zero, and **z** is a NaN.
- **fma(x, y, z)** returns a NaN and raises the invalid exception if one of **x** and **y** is infinite, the other is zero, and **z** is not a NaN.
- **fma(x, y, z)** returns a NaN and raises the invalid exception if **x** times **y** is an exact infinity and **z** is also an infinity but with the opposite sign.

*Include fma in the list of <math.h> (but not <complex.h>) functions that have a type-generic macro.*